Exercise 9: Facial Keypoints Detection

Keypoint Model

The __init__ function:

```
################################################################################

def conv_sandwich(inp, out, kernel_size, stride, pad):

    conv = nn.Conv2d(inp, out, kernel_size, stride, pad)
    nn.init.kaiming_normal_(conv.weight, nonlinearity="relu")

    return nn.Sequential(
        conv,
        nn.MaxPool2d(2, 2),
        nn.ReLU()
    )
```

Feature extraction

```
layers = []
layers.append(conv_sandwich(1, 32, kernel_size=3, stride=1, pad=1))
layers.append(conv_sandwich(32, 64, kernel_size=3, stride=1, pad=1))
layers.append(conv_sandwich(64, 128, kernel_size=3, stride=1, pad=1))
layers.append(conv_sandwich(128, 256, kernel_size=3, stride=1, pad=1))
self.convs = nn.Sequential(*layers)
```

Classification

```
self.fc1 = nn.Sequential(nn.Linear(256 * 6 * 6, 256), nn.ReLU())
self.fc2 = nn.Sequential(nn.Linear(256, 30))

nn.init.kaiming_normal_(self.fc1[0].weight, nonlinearity="relu")
nn.init.xavier_normal_(self.fc2[0].weight)
################################################################################
```

Tips:

- You can use nn.Sequential for stacking layers together in order to avoid writing this common block again.

- nn.Sequential doesn't take list as argument, so we need to decompose It by using the * operator.

Keypoint Model

```
def conv_sandwich(inp, out, kernel_size, stride, pad):
    return nn.Sequential(
        nn.Conv2d(inp, out, kernel_size, stride, pad),
        nn.MaxPool2d(2, 2),
        nn.ReLU()
    )

layers = []
layers.append(conv_sandwich(1, 32, kernel_size=3, stride=1, pad=1))
layers.append(conv_sandwich(32, 64, kernel_size=3, stride=1, pad=1))
layers.append(conv_sandwich(64, 128, kernel_size=3, stride=1, pad=1))
layers.append(conv_sandwich(128, 256, kernel_size=3, stride=1, pad=1))
self.convs = nn.Sequential(*layers)
```

Feature extraction

CONV2D

```
CLASS torch.nn.Conv2d(in_channels: int, out_channels: int, kernel_size: Union[T, Tuple[T,
      T]], stride: Union[T, Tuple[T, T]] = 1, padding: Union[T, Tuple[T, T]] = 0,
      dilation: Union[T, Tuple[T, T]] = 1, groups: int = 1, bias: bool = True,
      padding_mode: str = 'zeros')
```

[SOURCE]

For the first sandwich layer:

```
conv_sandwich(1, 32, kernel_size=3, stride=1, pad=1)
```

```
nn.Conv2d(inp, out, kernel_size, stride, pad),
```

after the conv2d:

*the output size = (width +2*padding– kernel_size)/stride + 1*

= (96+2-3)/1+1 = 96

```
nn.MaxPool2d(2, 2),
```

after maxpooling: 96/2 = 48

output dimension: (32,48,48)

After 4 sandwich layers, the output dimension is (256,6,6)

Keypoint Model - forward

```
def forward(self, x):

    # check dimensions to use show_keypoint_predictions later
    if x.dim() == 3:
        x = torch.unsqueeze(x, 0)
    ########################################################################
    # TODO: Define the forward pass behavior of your model                 #
    # for an input image x, forward(x) should return the                   #
    # corresponding predicted keypoints.                                   #
    # NOTE: what is the required output size?                              #
    ########################################################################

    x = self.convs(x)
    x = x.view(x.size(0), -1)
    x = self.fc1(x)
    x = self.fc2(x)

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return x
```

Remark:
Keep in mind that we need to reshape the output after applying the convolutional layers.

Training Loop

```python
########################################################################
# TODO - Train Your Model                                              #
########################################################################


import torch.optim as optim
from torch import nn

batch_size = 20
n_epochs = 2

criterion = nn.MSELoss()
train_loader = DataLoader(
    train_dataset,
    batch_size=batch_size,
    shuffle=True,
    num_workers=0,
)

optimizer = optim.SGD(
    model.parameters(),
    lr=0.01,
    momentum=0.9,
    weight_decay=1e-6,
    nesterov=True
)
```

```python
model.to(device)
model.train()  # prepare net for training
running_loss = 0.0
avg_loss = 0.0
for epoch in range(n_epochs):
    for i, data in enumerate(train_loader):
        image, keypoints = data["image"].to(device), data["keypoints"].to(device)
        predicted_keypoints = model(image).view(-1, 15, 2)
        loss = criterion(torch.squeeze(keypoints), torch.squeeze(predicted_keypoints))
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        running_loss += loss.item() if abs(loss.item() - avg_loss) < 100 else 0
        if i % 10 == 9:  # print every 10 batches
            avg_loss = running_loss / (len(train_loader) * epoch + i)
            print(
                "Epoch: {}, Batch: {}, Avg. Loss: {}".format(epoch + 1, i + 1, avg_loss)
            )
print("Finished Training")

########################################################################
#                           END OF YOUR CODE                           #
########################################################################
```

- Load training data in batches and shuffle the data with PyTorch's DataLoader class.
- Train the model and track the loss

Hyperparameters tuning:

We have trained the model with different combination of hyperparameters, and the best Score we have achieved is **283**.
You can train with other hyperparameters and get better results!

```
layers.append(conv_sandwich(1, 32, kernel_size=3, stride=1, pad=1))
layers.append(conv_sandwich(32, 256, kernel_size=3, stride=1, pad=1))
layers.append(conv_sandwich(256, 128, kernel_size=3, stride=1, pad=1))
layers.append(conv_sandwich(128, 256, kernel_size=3, stride=1, pad=1))
self.convs = nn.Sequential(*layers)
self.fc1 = nn.Sequential(nn.Linear(256 * 6 * 6, 256), nn.ReLU())
self.fc2 = nn.Sequential(nn.Linear(256, 30), nn.Tanh())
```

weight decay	momentum	Learning rate	Score
1e-7	1.0	0.01	167.84
1e-7	0.9	0.01	163.06
1e-6	1.0	0.01	127.22
1e-6	0.9	0.01	156.90
1e-7	1.0	0.1	0.94
1e-7	0.9	0.1	251.66
1e-6	1.0	0.1	0.99
1e-6	0.9	0.1	121.56

```
layers.append(conv_sandwich(1, 32, kernel_size=3, stride=1, pad=1))
layers.append(conv_sandwich(32, 64, kernel_size=3, stride=1, pad=1))
layers.append(conv_sandwich(64, 128, kernel_size=3, stride=1, pad=1))
layers.append(conv_sandwich(128, 256, kernel_size=3, stride=1, pad=1))
self.convs = nn.Sequential(*layers)
self.fc1 = nn.Sequential(nn.Linear(256 * 6 * 6, 256), nn.ReLU())
self.fc2 = nn.Sequential(nn.Linear(256, 30), nn.Tanh())
```

weight decay	momentum	Learning rate	Score
1e-7	1.0	0.01	130.93
1e-7	0.9	0.01	160.91
1e-6	1.0	0.01	101.45
1e-6	0.9	0.01	160.06
1e-7	1.0	0.1	0.96
1e-7	0.9	0.1	259.32
1e-6	1.0	0.1	0.70
1e-6	0.9	0.1	283.31

Optional Exercise 9: Spatial Batch Normalization

The forward pass

```python
def spatial_batchnorm_forward(x, gamma, beta, bn_param):
    """
    ...
    """
    out, cache = None, None

    ########################################################################
    # TODO: Implement the forward pass for spatial batch normalization.    #
    #                                                                      #
    # HINT: You can implement spatial batch normalization using the        #
    # vanilla version of batch normalization defined above. Your           #
    # implementation should be very short; ours is less than five lines.   #
    ########################################################################

    # Computation in one sweep by rearranging the dims to fit into
    # the batchnorm_forward framework
    x_swapped = np.transpose(x, (0, 2, 3, 1))
    x_swapped_reshaped = np.reshape(x_swapped, (-1, x_swapped.shape[-1]))

    out_temp, cache = batchnorm_forward(
        x_swapped_reshaped, gamma, beta, bn_param)
    out = np.transpose(np.reshape(out_temp, x_swapped.shape), (0, 3, 1, 2))

    ########################################################################
    #                           END OF YOUR CODE                           #
    ########################################################################
    return out, cache
```

- Unlike the normal batchnorm which computes mean and variance of each feature, spatial batchnorm computes them of each channel.

- We only need to rearrange the dimensions of data and then use the normal batchnorm forward function here.

The backward pass

```python
def spatial_batchnorm_backward(dout, cache):
    """
    """
    dx, dgamma, dbeta = None, None, None

    ###########################################################################
    # TODO: Implement the backward pass for spatial batch normalization.     #
    #                                                                         #
    # HINT: You can implement spatial batch normalization using the          #
    # vanilla version of batch normalization defined above. Your             #
    # implementation should be very short; ours is less than five lines.     #
    ###########################################################################

    dout_swapped = np.transpose(dout, (0, 2, 3, 1))
    dout_swapped_reshaped = np.reshape(
        dout_swapped, (-1, dout_swapped.shape[-1]))

    dx_sr, dgamma, dbeta = batchnorm_backward(dout_swapped_reshaped, cache)

    dx = np.transpose(np.reshape(dx_sr, dout_swapped.shape), (0, 3, 1, 2))

    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return dx, dgamma, dbeta
```

- Similar as the forward pass, in the backward pass we can compute the gradients by using the backprop from normal batchnorm with the rearranged dimensions.

Questions? Piazza ☺